Exhibit B

```
// This file generates synthetic sinusoids in order to analyze the
characteristic of
// a mircophone.

#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>
#include <ctime>
#include <algorithm>
#include <functional>
#include <complex>
#include "audio/WavFile.H"
#include "audio/AudioBuffer.H"
#include "nsp.h"

//#define nsp_UsesLms
//#include <nsp.h>


#ifndef M_PI
#define M_PI            3.14159265358979323846
#endif


#define SAMPLEPERSECOND 44100
#define WINDOWSIZE 1024
// calculates the frequency resolution of audio signals
#define DELTAFREQUENCY (double(44100) / double(1024))
// convert sample number to time in seconds.
#define SAMPLE2TIME(s) (double(s) / double(SAMPLEPERSECOND))


using namespace std;



//
-------------------------------------------------------------------------
---------------
//   signal ---> o------ Delay --- + Sum ---+----- e
//           +-> o                    -     |
//           |                        |     |
//           |           |----------|  |     |
//   noise  -+-> o------+ Adaptive |--+     |
//                      | filter   |        |
//                      |-----/----|        |
//                           /               |
//                          |---------------+
//
-------------------------------------------------------------------------
---------------

const float STEP = 0.01f;
const int NTAPS = 2048;
```

```
const int DELAY = NTAPS / 2;


int
cancelNoise(string waveFileName)
{
        int n;
        float *taps = nspsMalloc( NTAPS );
        float *dlyl = nspsMalloc( NTAPS );
        float err = 0;


        ReadWaveFile wavFile(waveFileName.c_str());
        int noOfSamplesPerChannel = wavFile.getNoOfSamplesPerChannel();
        WriteWaveFile wavFileLeftRight("h:/leftright.wav",
wavFile.wf.nSamplesPerSec, true);
        cout << wavFile.wf.nSamplesPerSec << "\t" << noOfSamplesPerChannel
<< endl;


        clock_t startTime = clock();

        NSPLmsTapState tapStPtr;
        NSPLmsDlyState dlyStPtr;

        /// init taps delay line to zero
        for( n=0; n<NTAPS; ++n ) taps[n] = dlyl[n] = 0;

        ///  Initialize filter
        nspsLmslInit( NSP_LmsDefault, taps, NTAPS, STEP, 0.0f, 0, &tapStPtr
);
        /// Initialize delay line
        nspsLmslInitDlyl( &tapStPtr, dlyl, TRUE, &dlyStPtr );
        /// Filter LEN samples using single-rate adaptive filtering

        // each pass process (one+delta) seconds of samples
        float *left  = new float[noOfSamplesPerChannel];
        float *right = new float[noOfSamplesPerChannel];
        wavFile.getSamples(left, right, noOfSamplesPerChannel);

        for (int s=DELAY ; s<noOfSamplesPerChannel; s++) {
                float d2 = nspsLmsl( &tapStPtr, &dlyStPtr, left[s]/32768.0f,
err );
                float d  =  right[s-DELAY]/32768.0f;
                err = d - d2;

                wavFileLeftRight.addSample(err*32768.f,err*32768.f);
                //wavFileLeftRight.addSample(err,right[s-DELAY]);
                if (s%wavFile.wf.nSamplesPerSec == 0)
                        cout << s/wavFile.wf.nSamplesPerSec << endl;
        }


        clock_t stopTime = clock();
        cout << "Total processing time: " << (stopTime -
startTime)/CLOCKS_PER_SEC << endl;
```

```
        nspFree( dlyl );
        nspFree( taps );
        delete[] left; delete[] right;

        return 0;
}

//
-------------------------------------------------------------------------
---------------
// - calculate total perceptual loudness ("gesamtLautheit")
//
-------------------------------------------------------------------------
---------------
class ConvolveAudioBuffer {

public:
        static float* mask;
        static int maskSize;
        static int noOfSamplesProcessed;

public:
        ConvolveAudioBuffer() {}
        ~ConvolveAudioBuffer() {}


        float operator() (float* windowSamples, int windowSize, int
samplingRate, float sampleScale)
        {
                int windowSize2 = windowSize / 2;
                int windowSizeBase2 = 0;
                int temp = windowSize;
                while (temp > 1) { temp >>= 1; windowSizeBase2++; }
                float* result = new float[windowSize + maskSize];
                nspsConv(windowSamples, windowSize, mask, windowSize,
result);
                float resultValue = result[maskSize-1];
                delete result;

                noOfSamplesProcessed++;
                return resultValue;
        }
};

float* ConvolveAudioBuffer::mask=0;
int ConvolveAudioBuffer::maskSize=1024;
int ConvolveAudioBuffer::noOfSamplesProcessed=0;



//
-------------------------------------------------------------------------
---------------
// - write audio test file to test mic's frequency response
//
-------------------------------------------------------------------------
```

```
---------------
int
writeTestWaveFile(int argc, char** argv)
{
        WriteWaveFile wavFile("c:/temp/test.wav", SAMPLEPERSECOND);
        double frequency = DELTAFREQUENCY/2; // of the sinusoid
        double time = 0;

        int j=0;
        for (int loop=0 ; loop<WINDOWSIZE/2 ; loop++, frequency+=
DELTAFREQUENCY) {
                // cut of all frequencies above 15000
                if ((frequency-DELTAFREQUENCY/2) > 15000) break;

                cout << "Frequency: " << frequency << endl;

                for (int i=0 ; i<SAMPLEPERSECOND*4 ; i++, j++) {
                        time = SAMPLE2TIME(j);
                        wavFile.addSample(short(10000 *
sin(frequency*2*M_PI*time)) );
                }
                for (int i=0 ; i<SAMPLEPERSECOND ; i++, j++) {
                        wavFile.addSample(short(0));
                }
        }
        return 0;
}


//
-------------------------------------------------------------------------
---------------
// - get loudness of the left and right audio channel
//
-------------------------------------------------------------------------
---------------
bool
generateStatistics(int argc, char** argv)
{
        ReadWaveFile wavFile("d:/TestSignalForMicrophone.wav");
        int s = wavFile.getNoOfSamplesPerChannel();
        short left, right;

        clock_t startTime = clock();
        vpl::AudioBuffer<vpl::gesamtLautheit>  audiobuffer1;
        vpl::AudioBuffer<vpl::gesamtLautheit>  audiobuffer2;
        for (int i=0 ; i<wavFile.getNoOfSamplesPerChannel() ; i++) {
                wavFile.getSample(left, right);
                audiobuffer1.addSample(left);
                audiobuffer2.addSample(right);
                if (i%wavFile.wf.nSamplesPerSec == 0)
                        cout << i/wavFile.wf.nSamplesPerSec << "\t" << left
<< "\t" << right << endl;
        }
        clock_t stopTime = clock();
        cout << "Total processing time: " << (stopTime -
startTime)/CLOCKS_PER_SEC << endl;
```

```
        return 0;
}

//
-------------------------------------------------------------------------
--------------
// - get loudness of the left and right audio channel
//
-------------------------------------------------------------------------
--------------
bool
generateStatistics2(int argc, char** argv)
{
        const int WINDOWSIZE2 = WINDOWSIZE / 2;

        ReadWaveFile wavFile("d:/TestSignalForMicrophone.wav");
        int s = wavFile.getNoOfSamplesPerChannel();

        int windowSizeBase2 = 0;
        int temp = WINDOWSIZE;  while (temp > 1) { temp >>= 1;
windowSizeBase2++; }
        float left[WINDOWSIZE+2], right[WINDOWSIZE+2];


        //
-------------------------------------------------------------------------
        // CREATE CONVERSION MASK
        //
-------------------------------------------------------------------------
        const int MASKSIZE = 1024;
        const int MASKSIZE2 = MASKSIZE / 2;
        float maskD[MASKSIZE+2]; nspsbSet(0.0f, maskD, MASKSIZE+2);
        float maskMic[MASKSIZE+2]; nspsbSet(0.0f, maskMic, MASKSIZE+2);
        //
-------------------------------------------------------------------------


        clock_t startTime = clock();

        // read chunk at the beginning of wave file: 393*1024 samples =
9*44100+5532
        for (int i=0 ; i<393 ; i++) wavFile.getSamples(left, right,
WINDOWSIZE);


        FILE* testData = fopen("d:/mics.test","w");

        const int noOfSamples = 5 * 44100-183; // the length of one sinusoid
        int noOfHeadingWindows = 44100/2 / WINDOWSIZE; // skip the first
half second
        int noOfWindows = 44100*3/WINDOWSIZE - noOfHeadingWindows;
        int noOfTailWindows = noOfSamples/WINDOWSIZE - noOfWindows -
noOfHeadingWindows;
        int noOfTailWindowsRest = noOfSamples -
(noOfHeadingWindows+noOfWindows+noOfTailWindows)*WINDOWSIZE ;
```

```
        for (int i=0,j=0 ; j<222 && i<wavFile.getNoOfSamplesPerChannel() ;
i+=noOfSamples, j++) {
                cout << j << ": ";

                // process the current frequency
                complex<double> ratioSum = 0.0;
                int countNoOfRatios = 0;

                for (int x=0 ; x<noOfHeadingWindows ; x++)
wavFile.getSamples(left, right, WINDOWSIZE);

                for (int n=0 ; n<noOfWindows ; n++) {
                        wavFile.getSamples(left, right, WINDOWSIZE);

                        // calculate frequency magnitudes
                        nspsWinHamming(left,  WINDOWSIZE);
                        nspsRealFft(left, 10, NSP_Forw);
                        nspsWinHamming(right, WINDOWSIZE);
                        nspsRealFft(right, 10, NSP_Forw);


                        float windowMagnitudeL[WINDOWSIZE2+1];
                        nspcbMag((const SCplx*) left,  windowMagnitudeL,
WINDOWSIZE2+1);
                        float windowMagnitudeR[WINDOWSIZE2+1];
                        nspcbMag((const SCplx*) right, windowMagnitudeR,
WINDOWSIZE2+1);


                        // CREATE MICROPHONE STATISTIC
                        float* pmaxL = std::max_element(windowMagnitudeL,
windowMagnitudeL+WINDOWSIZE2);
                        float* pmaxR = std::max_element(windowMagnitudeR,
windowMagnitudeR+WINDOWSIZE2);
                        int imaxL = int(pmaxL - windowMagnitudeL);
                        int imaxR = int(pmaxR - windowMagnitudeR);

                        //float minv = *std::min_element(windowMagnitude,
windowMagnitude+windowSize2);
                        //fprintf(testData, "%d %d %f %f %f %f %f %f\n",
imaxL, imaxR,
                        //      *pmaxL, *pmaxR,
                        //      left[imaxL*2], left[imaxL*2+1],
right[imaxR*2], right[imaxR*2+1]);
                        complex<float> r(right[imaxR*2], right[imaxR*2+1]);
                        complex<float> l(left[imaxL*2], left[imaxL*2+1]);
                        complex<float> ratio = l/r;
                        if ((imaxL == j) && (imaxR == j)) {
                                ratioSum += ratio;
                                countNoOfRatios++;
                                //fprintf(testData, "%d %d %f %f %f %f\n",
imaxL, imaxR, *pmaxL, *pmaxR, ratio.real(), ratio.imag());

                        }
                }
                ratioSum /= countNoOfRatios;
                fprintf(testData, "%f %f\n", ratioSum.real(),
```

```
ratioSum.imag());
                maskD[j*2]   = ratioSum.real();
                maskD[j*2+1] = ratioSum.imag();

                for (int x=0 ; x<noOfTailWindows ; x++)
wavFile.getSamples(left, right, WINDOWSIZE);
                wavFile.getSamples(left, right, noOfTailWindowsRest);
        }

        fclose(testData);
        clock_t stopTime = clock();
        cout << "Total processing time: " << (stopTime -
startTime)/CLOCKS_PER_SEC << endl;




        //
-------------------------------------------------------------------------
        // CREATE CONVERSION MASK
        //
-------------------------------------------------------------------------
        // the first 3 frequencies are unsave
        for (int x=0; x<3 ; x++) {maskD[2*x]=1; maskD[2*x+1]=0;}
        for (int x=222 ; x<MASKSIZE2 ; x++) maskD[2*x]=1;
        ofstream outPut2("conversionMaskOrg.txt");
        copy(maskD, maskD+MASKSIZE, ostream_iterator<float>(outPut2, "\n"));
        outPut2.close();


        nspsCcsFft(maskD, 10, NSP_Inv);

        ofstream outPut("conversionMask.txt");
        for (int i=0 ; i<MASKSIZE ; i++) { maskMic[(i+MASKSIZE2-1) %
MASKSIZE] = maskD[i]; }
        maskMic[MASKSIZE-1] = 0;
        copy(maskMic, maskMic+MASKSIZE-1, ostream_iterator<float>(outPut,
"\n"));

        // determine the cut-off coefficients
/*      float* posIn  = find_if (maskMic, maskMic+MASKSIZE,
bind2nd(greater<float>(),0.01f));
        int usedMaskSize1 = posIn - maskMic;
        int usedMaskSize3 = MASKSIZE-1 - usedMaskSize1 -1;

        for (int i=0 ; i<MASKSIZE-usedMaskSize1 ; i++) { maskMic[i] =
maskMic[i+usedMaskSize1];}
        int usedMaskSize = usedMaskSize3 - usedMaskSize1 + 1;
        copy(maskMic, maskMic+usedMaskSize, ostream_iterator<float>(outPut,
"\n"));
*/      outPut.flush();
```

```
        return 0;
}


//
-------------------------------------------------------------------------
--------------
// - correlate the response of the 2 mics to the sinusoid of a certain
frequency
//
-------------------------------------------------------------------------
--------------
bool
correlateChannels()
{
        ReadWaveFile wavFile("d:/TestSignalForMicrophone.wav");
        int s = wavFile.getNoOfSamplesPerChannel();
        const int noOfSamples = 5 * 44100-183; // the length of one sinusoid
        float* left  = new float[noOfSamples];
        float* right = new float[noOfSamples];

        clock_t startTime = clock();
        // read chunk at the beginning of wave file: 393*1024 samples =
9*44100+5532
        wavFile.getSamples(left, right, 3 * 44100);
        wavFile.getSamples(left, right, 3 * 44100);
        wavFile.getSamples(left, right, 3 * 44100);
        wavFile.getSamples(left, right, 5532);

        wavFile.getSamples(left, right, noOfSamples);
        for (int i=0,j=1 ; j<222 && i<wavFile.getNoOfSamplesPerChannel() ;
i+=noOfSamples, j++) {
                cout << j << ": ";
                // REMEMBER: cut of first half second and last half second
                // find best correlation
                wavFile.getSamples(left, right, noOfSamples);
                char name[1000]; sprintf(name,"d:/%03d.wav", j);
                WriteWaveFile wavFreq(name, SAMPLEPERSECOND, true);
                wavFreq.addSamples(left, right, noOfSamples);
                //avFreq.addSamples(left+44100/2, right+44100/2, 2*44100);

                for (int n=0 ; n<noOfSamples ; n++) left[n] = -left[n];
                const float stepFrequency = float(44100) / float(1024);
                int noOfSamplesPerWave = int(44100 / (j * stepFrequency +
stepFrequency/2.0f));
                cout << noOfSamplesPerWave << "\t";
                float *result = new float[201]; //[noOfSamplesPerWave+1];
                //nspsCrossCorr(left+44100/2, 3*44100, right+44100/2,
3*44100, result, 0, noOfSamplesPerWave);
                nspsCrossCorr(left+44100/2, 2*44100, right+44100/2, 2*44100,
result, 0, 200);
                //float* minPos = min_element(result,
result+noOfSamplesPerWave+1);

                // return local maxs
                for (int n=1 ; n<200-1 ; n++) {
                        if ((result[n-1] <= result[n]) && (result[n+1] <=
```

```
result[n]))
                                cout << n << "\t";
                }
                delete result;
                cout << endl;
        }
        clock_t stopTime = clock();
        cout << "Total processing time: " << (stopTime -
startTime)/CLOCKS_PER_SEC << endl;

        return 0;
}



//
------------------------------------------------------------------------
---------------
// - diente dazu, Probleme mit der FFT auszuraeumen
//
------------------------------------------------------------------------
---------------
void
generateFFTTestPatterns(int argc, char** argv)
{
        const int MASKSIZE = 1024;
        const int MASKSIZE2 = MASKSIZE / 2;
        float maskD[MASKSIZE+2]; nspsbSet(0.0f, maskD, MASKSIZE+2);

        ofstream outPutOrg("conversionMaskOrg.txt");
        nspsbSet(0.0f, maskD, MASKSIZE+2);
        for (int i=0 ; i<MASKSIZE2 ; i++) {
                float iFloat = float(i) / float (MASKSIZE2);
                maskD[i] = exp(-iFloat*iFloat / (2 * 0.0001));
                maskD[MASKSIZE-1-i] = exp(-iFloat*iFloat / (2 * 0.0001));
        }
        for (int i=0 ; i<MASKSIZE ; i++) {
                //outPutOrg << maskD[i] << endl;
        }


        nspsRealFft(maskD, 10, NSP_Forw);
        //float mag[MASKSIZE2+2]; nspsbSet(0.0f, mag, MASKSIZE2+2);
        //nspcbMag((const SCplx*)maskD, mag, 512);
        for (int i=0 ; i<MASKSIZE ; i++) {
                outPutOrg << maskD[i] << endl;
                //outPutOrg << mag[i] << endl;
        }
}

//
------------------------------------------------------------------------
---------------
// - diente dazu, Probleme mit der FFT auszuraeumen
//
------------------------------------------------------------------------
---------------
```

```
void
generateFFTTestPatterns2(int argc, char** argv)
{
        const int MASKSIZE = 1024;
        const int MASKSIZE2 = MASKSIZE / 2;
        float maskD[MASKSIZE+2]; nspsbSet(0.0f, maskD, MASKSIZE+2);
        for (int i=0 ; i<MASKSIZE2 ; i++) maskD[2*i] = 1.0f;


        ofstream outPutOrg("conversionMaskOrg.txt");
        nspsCcsFft(maskD, 10, NSP_Inv);
        for (int i=0 ; i<MASKSIZE ; i++)
                outPutOrg << maskD[i] << endl;

}



//
------------------------------------------------------------------------
// - Online Training
------------------------------------------------------
//
------------------------------------------------------------------------
void
onlineTraining(string waveFileName)
{
        ReadWaveFile wavFile(waveFileName.c_str());
        int noOfSamplesPerChannel = wavFile.getNoOfSamplesPerChannel();


        WriteWaveFile wavFileLeftRight("c:/temp/leftright.wav",
SAMPLEPERSECOND, true);

        clock_t startTime = clock();

        const int NOOFSAMPLES = 1024;
        const int NOOFSAMPLES2= NOOFSAMPLES/2;
        const int POWEROFSAMPLES = 10;
        const float threshold = 100 * NOOFSAMPLES;
        float   left  [NOOFSAMPLES+2];  float leftMag [NOOFSAMPLES2+1];
        float   right [NOOFSAMPLES+2];  float rightMag[NOOFSAMPLES2+1];
        float   ratio [NOOFSAMPLES+2];  nspsbZero(ratio, NOOFSAMPLES+2);
        int     count [NOOFSAMPLES2+1]; memset(count, 0, sizeof(int) *
(NOOFSAMPLES2+1));

        // each pass process (one+delta) seconds of samples
        for (int s=0 ; s<noOfSamplesPerChannel/NOOFSAMPLES ; s++) {
                wavFile.getSamples(left, right, NOOFSAMPLES);

                nspsRealFft(left, POWEROFSAMPLES, NSP_Forw);
                nspsRealFft(right, POWEROFSAMPLES, NSP_Forw);
                nspcbMag((const SCplx *)left , leftMag,  NOOFSAMPLES/2+1);
                nspcbMag((const SCplx *)right, rightMag, NOOFSAMPLES/2+1);

                float leftMean  = nspsMean(leftMag,  NOOFSAMPLES2+1);
```

```
                float rightMean = nspsMean(rightMag, NOOFSAMPLES2+1);
                for (int i=0 ; i<NOOFSAMPLES2+1 ; i++) {
                        if ((leftMag[i] > threshold) && (rightMag[i] >
threshold)) {
                                // cout << i << " " ;
                                count[i]++;
                                complex<float> leftC (left [2*i], left
[2*i+1]);
                                complex<float> rightC(right[2*i],
right[2*i+1]);
                                complex<float> ratioC = leftC / rightC;
                                ratio[2*i] += ratioC.real();
                                ratio[2*i+1] += ratioC.imag();
                        }
                }
                cout << (s*NOOFSAMPLES)/SAMPLEPERSECOND << "\t" << leftMean
<< "\t" << rightMean << endl;
        }

        // Calculate deconvolution mask
        for (int i=0 ; i<NOOFSAMPLES2+1 ; i++) {
                if (count[i]) {
                        ratio[2*i] /= count[i];
                        ratio[2*i+1] /= count[i];
                } else {
                        ratio[i] = 0.5;
                        ratio[2*i+1] = 0;
                }
        }


        nspsCcsFft(ratio, POWEROFSAMPLES, NSP_Inv);


        ofstream outPut("conversionMask.txt");
        float   maskMic [NOOFSAMPLES];
        for (int i=0 ; i<NOOFSAMPLES ; i++) { maskMic[(i+NOOFSAMPLES2-1) %
NOOFSAMPLES] = ratio[i]; }
        copy(maskMic, maskMic+NOOFSAMPLES, ostream_iterator<float>(outPut,
"\n"));
        outPut.flush();


        clock_t stopTime = clock();
        cout << "Total processing time: " << (stopTime -
startTime)/CLOCKS_PER_SEC << endl;
}




//
------------------------------------------------------------------------
// - MAIN
------------------------------------------------------------------
//
```

```
int
main(int argc, char** argv)
{
        cancelNoise("h:/OpenHouse.wav");
        //generateStatistics(argc, argv);
        return 0;
        //string wavInputFileName = "d:/shortWave.wav";
        string wavInputFileName = "d:/cap2.wav";
        onlineTraining(wavInputFileName);

        //
------------------------------------------------------------------------
        // CREATE CONVERSION MASK
        //
------------------------------------------------------------------------
        const int MASKSIZE = 1024;
        const int MASKSIZE2 = MASKSIZE / 2;

        float maskD[MASKSIZE+2]; nspsbSet(0.0f, maskD, MASKSIZE+2);
        float maskMic[MASKSIZE+2]; nspsbSet(0.0f, maskMic, MASKSIZE+2);
        /*
        ifstream inPut("conversionMaskOrg.txt");
        assert(inPut != 0);
        for (int i=0 ; i<MASKSIZE ; i++) {
                if (!inPut.eof())
                        inPut >>  maskD[i];
        }


        nspsCcsFft(maskD, 10, NSP_Inv);
        ofstream outPut("conversionMask.txt");
        for (int i=0 ; i<MASKSIZE ; i++) { maskMic[(i+MASKSIZE2-1) %
MASKSIZE] = maskD[i]; }
        maskMic[MASKSIZE-1] = 0;
        */

        ifstream inPut("conversionMask.txt");
        copy(istream_iterator<float>(inPut), istream_iterator<float>(),
maskMic);

        // determine the cut-off coefficients
        /*
        float* posIn  = find_if (maskMic, maskMic+MASKSIZE,
bind2nd(greater<float>(),0.01f));
        int usedMaskSize1 = posIn - maskMic;
        int usedMaskSize3 = MASKSIZE-1 - usedMaskSize1 -1;

        for (int i=0 ; i<MASKSIZE-usedMaskSize1 ; i++) { maskMic[i] =
maskMic[i+usedMaskSize1];}
        int usedMaskSize = usedMaskSize3 - usedMaskSize1 + 1;
        copy(maskMic, maskMic+usedMaskSize, ostream_iterator<float>(outPut,
"\n"));
        outPut.flush();*/
        for (int i=0 ; i<1024/*usedMaskSize*/ ; i++) {
                cout << maskMic[i] << endl;
        }
```

```
        int usedMaskSize = 1024;
        ConvolveAudioBuffer::mask=maskMic;
        ConvolveAudioBuffer::maskSize=usedMaskSize;



        ReadWaveFile wavFile(wavInputFileName.c_str());
        //ReadWaveFile wavFile("d:/TestSignalForMicrophone.wav");
        int s = wavFile.getNoOfSamplesPerChannel();
        int noOfSamplesPerChannel = wavFile.getNoOfSamplesPerChannel();
        WriteWaveFile wavFileLeftRight("c:/temp/leftright.wav",
SAMPLEPERSECOND, true);


        clock_t startTime = clock();

        float left  [SAMPLEPERSECOND + MASKSIZE];
        float right [SAMPLEPERSECOND + MASKSIZE];
        float result[SAMPLEPERSECOND + 2 * MASKSIZE];

        // prefill buffers
        wavFile.getSamples(left, right, MASKSIZE);

        // each pass process (one+delta) seconds of samples
        for (int s=0 ; s<10 ; s++) {
                wavFile.getSamples(left + MASKSIZE, right + MASKSIZE,
SAMPLEPERSECOND);
                nspsConv(right, SAMPLEPERSECOND+MASKSIZE, maskMic, MASKSIZE,
result);
                //nspsbMpy1(1.2f,  right+MASKSIZE/2, SAMPLEPERSECOND+1);
                //nspsbSub2(left+MASKSIZE/2, result+MASKSIZE-1,
SAMPLEPERSECOND+1);
                wavFileLeftRight.addSamples(left+MASKSIZE/2,
result+MASKSIZE-1, SAMPLEPERSECOND+1);
                //nspsbMpy1(1.0f/1.2f,  right+MASKSIZE/2,
SAMPLEPERSECOND+1);

                memcpy(right, right+SAMPLEPERSECOND, sizeof(float) *
MASKSIZE);
                memcpy(left,  left +SAMPLEPERSECOND, sizeof(float) *
MASKSIZE);
                cout << s <<  endl;
        }

        clock_t stopTime = clock();
        cout << "Total processing time: " << (stopTime -
startTime)/CLOCKS_PER_SEC << endl;



        return 0;
}
```